quwsarohi@gmail.com Section: String Processing Page : 1

**// String Matching**

**// Knuth Morris Pratt**

**// Complexity : O(String + Token)**

char P[2000010], T[1000010]; //T is the string that we need to find

int P\_Size, T\_Size, Table[1000010]; //S is the string in which we have to find

void PrefixTable() { // Builds the prefix table

int i = 0, j = -1; // Table contains the prefix table

Table[0] = -1;

while(i < T\_Size) { // Pre-process the pattern string T

while(j >= 0 && T[i] != T[j]) // If different, reset j using Table

j = Table[j]; // j = last point where i’th element = j’th element

i++, j++; // If same, advance both pointers

Table[i] = j; //Prefix Table

} } 0 1 2 3 4 5 6 7 8 9 0 1 2 3

P = S E V E N T Y S E V E N

int KmpSearch() { Table = -1 0 0 0 0 0 0 0 0 1 2 3 4 5

register int i = 0, j = 0, cnt = 0;

while(i < P\_Size) {

while(j >= 0 && P[i] != T[j]) // Search through string P

j = Table[j]; // If different, reset j using T

i++, j++; // if same, advance both pointers

if(j == T\_Size) { //the **match found** in i-j, if i-j = 0, then the whole string is matched

cnt++; // This happens when the string is equal in length of the token

//printf("%d'th Match found at %d\n", cnt, i-j); //the leftmost index

j = Table[j]; //j contains the first segment index that is matched in token

} }

return cnt; // Return the number of successful matches

**}**

**1 2 3 4 5**

012 3 4 5 6 789012 3456789012345678 9 0 12345678 90 1 2 3 4 5 6 7 8 9 0

P = I DO NOT LIKE SEVENTY SEV BUT SEVENTY SEVENTY SEVEN

T = SEVENTY SEVEN

01 2 3 4 567890 1 2

SEVENTY SEVEN

01 2 3 4 56 7 890 1 2

**//Trie, LRS, LCS**

**//Complexity : Build : O(S), Search : O(S)**

struct node {

//int visited; // Add if repeated substring needed

bool isEnd; // Indicates if this node contains a string that ends at this character

node \*next[11]; // How many child a root/parent node may contain

node() { // Initializer

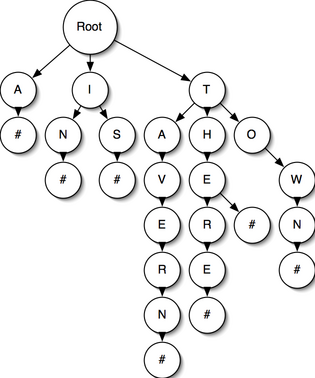
isEnd = false;

for(int i = 0; i < 10; i++)

next[i] = NULL;

}};

quwsarohi@gmail.com Section: String Processing Page : 2

bool create(char str[], int len, node \*current) {

for(int i = 0; i < len; i++) {

int pos = str[i] - '0';

if(current->next[pos] == NULL) // If this point don’t have child

current->next[pos] = new node(); // Initialize child

current = current→next[pos];

current→visited++; // Use this line if number of times visited in a node is

} //required

current->isEnd = true;

return false;

}

void del(node \*current) { // Deletes Trie

for(int i = 0; i < 10; i++)

if(current->next[i] != NULL)

del(current→next[i]); Fig: Trie, # are isEnd = True

delete current;

}

bool found = 0;

void search(node \*current) {

for(int i = 0; i < 10; i++) {

if(current->next[i] != NULL)

check(current->next[i]);

}

if(found) return;

if(current->isEnd && !found) {

for(int i = 0; i < 10 && !found; i++)

if(current->next[i] != NULL) {

found = 1;

} } }

string LongestRrepeatedSubstr(node \*current, string past) { // Longest Repeated String

int pos; // string past contains the past matched part

string longestRepeated; // y will contain the best repeated string longest and repeated

longestRepeated += past;

for(int i = 0; i < 4; i++) { **// Here every node contains four (4) child**

if(current->next[i] != NULL) { **// Change this line according to child**

if(current->next[i]->visited > 1) { // If this segment/char is visited more than once

string tmp;

tmp += map\_to\_str[i]; // Take this str as tmp

tmp += LRS(current->next[i], ""); // Find next LRS

if(tmp.size() > y.size()) // If the string found in this node is larger than previous found string

longestRepeated = tmp; // Take the largest

else if(tmp.size() == y.size()) // If both found in this search and the previous

if(tmp < longestRepeated) // If tmp is lexicographically smaller

longestRepeated = tmp; // take tmp as repeated substring

} } }

return longestRepeated; // LRS of ‘AG**AG**AG’ is 2, ‘AG**AG**’ and ‘**AG**AG’ both **AG** is common

}

// For two string Longest Common Substring is the longest substring that is the node is visited by two or more strings

// This code is for two LCS in two strings

int max\_len = -1;

vector<string>lcs\_str; //This contains all the substring

void longestCommonSubstrring(string past, node \*current) { // In struct add bitset<num\_of\_str>visited

for(int i = 0; i < 4; i++) { **// Here every node contains four (4) child**

if(current->next[i] != NULL) { **// Change this line according to child**

if(current->next[i]->visited.count() == 2) { // If the node is visited from both strings

string tmp;

tmp += past; // Take past string + new found string

tmp += map\_str\_to\_int[i];

max\_len = max(max\_len, (int)tmp.size()); // Find the maximum length string

LCS(tmp, current - > next[i]); // Go for deeper match, this will add the deepr strings before this substring

if(tmp.size() == mx\_len) // If This substring is the longest

lsc\_str.push\_back(tmp); // push to lcs\_str

} } } }

main() { ………...

node\* root = new node(); // Creating root node

for(int i = 0; i < string\_len; i++) // Use this to build Prefix Trie

create(Str+i, string\_len-i, 0, root); // Both LCS and LRS will need this

//Longest Repeated Substring

string lrs = LongestRrepeatedSubstr(root, “”); // Gives the longest repeated substring

node \*current = root;

for(int i = 0; i < y.size(); i++) {

if(current - > next[lrs[i]-’0’] != NULL) // lrs[i] – ‘0’ = index of that string

current = current→next[lrs[i]- ‘0’];

}

printf("%d\n", current - >visited); // In struct the visited int indicates how many times the node is visited

//Longest Common Substring

mx\_len = -1;

longestCommonSubstrring("", root);

for(int i = 0; i < lcs\_str.size(); i++)

if(lcs\_str[i].size() == mx\_len) //Only Printing the Longest Substring

printf("%s\n", lcs\_str[i].c\_str());

del(root);

}